Documentation for queue.h and queue.c

Steven Andrews, © 2003
See the document "LibDoc" for general information about this and other libraries.

```
typedef struct qstruct{
       void **x;
       void **k;
       int (*keycmp)(void *,void *);
       int n;
       int f;
       int b; } *queue;

#define qfrontkey(q) ((q)->b==(q)->f?0:(q)->k[(q)->f])

queue qalloc(int n,int (*keycmp)(void *,void *));
void qfree(queue q,int freek,int freex);
void qnull(queue q);
int enqueue(void *k,void *x,queue q);
int qpush(void *k,void *x,queue q);
int qinsert(void *k,void *x,queue q);
void *qfront(queue q,void **kptr);
void *qpop(queue q,void **kptr);
int qlength(queue q);
int qmaxlength(queue q);
int qnext(int i,void **kptr,void **xptr,queue q);
```

Requires: `stdlib.h, queue.h`

Example program: `LibTest.c`

Written 4/16/95, modified 4/20/95. Mostly rewritten 1/15/02 using CodeWarrior C. Some testing. Ported to Linux 1/16/02. Moved `FltCmp` to VoidComp.c 3/14/02. Added `qmaxlength` and `qnext` 3/7/03.

This implements a queue or stack of user definable length for arbitrary pointers using a circular array. It is also possible to maintain a sorted queue. While speed is slightly sacrificed, these routines are reasonably friendly about full and empty queues. When a queue is full, a new item overwrites the oldest item and the queue stays full. Reading from an empty queue results in a `NULL` returned value and the queue stays empty. The only significant thing to watch out for is that overwritten information is not freed, meaning that the data should not need freeing, the queue should not be the master list of data, or overfilling a queue should be avoided.

A queue actually contains two lists, one of items and one of keys for the items, although either (or both) may be `NULL`. An unsorted queue will typically make all keys `NULL` and ignore them, while a sorted queue might use just keys or might use both members. Since keys and items are type `void *`, they are maximally versatile and can be used directly for integers or characters, or as strings, or as pointers to structures.

The members of the `queue` structure include `x` and `k`, which are the list of items and a corresponding list of keys. `keycmp` is a function for comparing keys, and is only necessary for a sorted queue. Several simple key comparison routines

include FltCmp, StringCmp, and IntCmp in the VoidComp.c library. n, f, and b are queue indicies: n is the total number of spaces in the list, including usable spaces and one additional space so full lists can be distinguished from empty lists, f is the address of the front of the list, and b is one more than the address of the last element (a new element is added at position b). While it shouldn't generally be necessary, here is a useful code fragment for looking at each item of a queue, from front to back:

```
for(i=q->f;i!=q->b;i=(i+1)%q->n) x=q->x[i];
```

qfrontkey is written as a macro to make it fairly fast routine. If the queue is empty it returns NULL, otherwise it returns the key of the item at the front. The queue is not affected.

qalloc creates a new queue with n usable spaces and initializes all spaces to NULLs. keycmp is required for sorted queues and is sent in as NULL for unsorted ones. The return value is a pointer to a queue unless memory allocation failed, in which case NULL is returned.

qfree frees a queue. If freestuff is 1, any items and keys are freed as well. This routine is robust in that it only de-references and frees pointers that are not NULL.

qNULL sets the indicies to indicate an empty queue. It does not free any memory or erase the contents of cells.

enqueue adds an item and a key to the back of the line, used for FILO lists. It returns the total number of usable spaces remaining in the queue, or –1 if the routine just overwrote an old element.

qpush adds an item and a key to the front of the line, used for FIFO lists. As with enqueue, it returns the number of remaining spaces or –1.

qinsert inserts an item in the correct place of a sorted queue, where the keys are in ascending order from front to back. For this routine, keycmp needs to have been defined during allocation and the keys of the list members should all be valid. If an element is inserted to an already full queue, the item at the back of the list is dropped from the queue. As with enqueue, it returns the number of remaining spaces or –1.

qfront returns at the item at the front, but doesn't remove it from the list. If kptr is sent in as NULL, it is ignored; otherwise its value is set to the key at the front of the list. If the queue is empty, both returned values are NULL.

qpop is just like qfront, but it also removes the frontmost item and key.

qlength returns the number of items stored in the queue. If the queue is full, qlength returns 0, since it can't distinguish a full queue from an empty one.

qmaxlength returns the maximum number of items that can be stored in this queue, which is just q->n-1.

qnext is a useful command for looking at each item in the queue without changing the queue. i is the index of the previous item in the queue, or is -1 to indicate that the next item should be the first one. The return value is the index of the next item, along with pointers to the key and item stored there. Here is a typical use for this command:

```
i=-1;
while((i=qnext(i,&k,&x,q))>=0) { print out k and x }
```

If the key and/or item are not wanted, then pass a NULL pointer into kptr or xptr.